(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 006 440 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
07.06.2000 Bulletin 2000/23

(51) Int Cl.[7]: G06F 9/44

(21) Application number: 99309535.5

(22) Date of filing: 29.11.1999

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 30.11.1998 US 203043

(71) Applicant: SUN MICROSYSTEMS, INC.
Palo Alto, California 94043 (US)

(72) Inventor: Kanungo, Rajesh
Sunnyvale, California 94087 (US)

(74) Representative: Browne, Robin Forsythe, Dr.
Urquhart-Dykes & Lord
Tower House
Merrion Way
Leeds LS2 8PA West Yorkshire (GB)

(54) Display widget interaction in embedded systems using child graphics contexts

(57) A method and apparatus implementing a separate child context for each applet (or similar element) of a browser. A described embodiment of the present invention provides one or more child contexts that correspond to elements in the HTML for a web page displayed by a browser. For example, each applet executed by the browser has a corresponding and separate child context Each child context has an associated memory buffer. The browser also has a parent context, which each child context points to. When a graphic is displayed via a widget, the widget draws the graphic (such as a panel or a non-pressed button) in the child context of the applet and sets a "damage" flag in the child context. When the browser performs its main browser loop, it checks the status of the damaged flag for each element (including each applet). If the browser finds a damage flag that is set, this means that something was written into the child buffer and that the parent buffer needs updating. In this case, the browser "pulls" the information from the child buffer into the parent buffer, which is then used to update the display screen. Other components, called reactive components, present special problems and are treated specially. Reactive components are drawn directly into both the child and parent contexts and buffers without waiting for the main browser loop.

EP 1 006 440 A2

Description

FIELD OF THE INVENTION

[0001] This application relates to a method and apparatus for a user interface control and, specifically, to a method and apparatus that allows for fast processing of reactive components in a user interface.

BACKGROUND OF THE INVENTION

[0002] Systems that have embedded programs, such as certain information appliances and consumer products, often do not contain general purpose windowing software. In these systems, all graphics are generally done via calls to the built-in graphics routines, such as draw-line, draw rectangle, etc. In many conventional systems, graphics are drawn via interaction with the browser. Thus, if multiple applets are executing as different threads in the browser, they will experience a lack of concurrency when performing graphics operations.

SUMMARY OF THE INVENTION

[0003] A described embodiment of the present invention implements a separate child context for each applet (or similar browser element). A described embodiment of the present invention provides one or more child contexts that correspond to elements in the HTML for a web page displayed by a browser. For example, each applet executed by the browser has a corresponding and separate child context. Each child context has an associated memory buffer. The browser also has a parent context, which each child context points to. When a graphic is displayed via a widget, the widget draws the graphic (such as a panel or a non-pressed button) in the child context of the applet and sets a "damage" flag in the child context. When the browser performs its main browser loop, it checks the status of the damaged flag for each element (including each applet). If the browser finds a damage flag that is set, this means that something was written into the child buffer and that the parent buffer needs updating. In this case, the browser "pulls" the information from the child buffer into the parent buffer, which is then used to update the display screen. Thus, several separate threads in the JVM can be executing concurrently and updating graphics elements without interfering with each other.

[0004] The invention extends the AGL (Applications graphics library) routines to include (at least) routines to set up, delete, read, and modify child contexts and to copy information from child contexts to parent contexts. The DWL (developer's widget library) routines are extended to include (at least) a parameter identifying the context currently being worked on. The browser has been extended to implement the functionality of checking the damage flag of the context for each applet and pulling information from a buffer of a child context into the buffer of the parent context.

[0005] Reactive components present special problems. Reactive components are defined as components that must display interactive behavior. For example, when a user clicks on a button on the display screen, the button is momentarily displayed as "pressed" and soon thereafter displayed as "released." Such interactive behavior gives visual feedback to the user. Examples of reactive components include buttons, lists, choice bars, etc. present special problems. When a widget, for example, a button, needs to show interactive behavior (such as going up and down when pressed), this behavior must occur very rapidly - often too rapidly to necessarily be picked up by a main browser loop, which executes relatively infrequently. For example, in the described embodiment, the main browser loop executes approximately every 200 milliseconds (i.e., approximately five times per second). In contrast, a "down" button press followed by an "up" takes approximately 50 milliseconds. If the system were to wait for the browser loop, the button press would have to last much longer.

[0006] In accordance with the purpose of the invention, as embodied and broadly described herein, the invention relates to a method of displaying a reactive graphics element, comprising: sending a reactive graphics event, by a browser to a virtual execution environment; providing a child context in the virtual execution environment, corresponding to a parent context in the browser, where the child context corresponds to a current applet being executing by the browser; writing, by the virtual execution environment, a first graphic into a buffer of the child context and a buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen; waiting a predetermined amount of time; writing, by the virtual execution environment, a second graphic into the buffer of the child context and the buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen; and returning execution control to the browser from the virtual execution environment.

[0007] A fuller understanding of the invention will become apparent and appreciated by referring to the following description and claims taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and, together with the description, serve to explain the principles of the invention.

[0009] Fig. 1 (a) is a block diagram of a data processing system in accordance with one embodiment of the present invention.

[0010] Fig. 1 (b) is a block diagram of a data processing system in accordance with one embodiment of the

present invention.

[0011] Fig. 2 shows an example of a set top box system, including a remote unit, and a video display.

[0012] Fig. 3(a) is a block diagram of parent and child context data structures in accordance with an embodiment of the present invention.

[0013] Fig. 3(b) is a diagram showing the relationships between a browser, exemplary browser elements, and exemplary components of a specific browser element.

[0014] Fig. 4 is a block diagram of an orphan data structure in accordance with an embodiment of the present invention.

[0015] Fig. 5 is a block diagram showing a main browser loop and a JVM loop.

[0016] Fig. 6 is a flowchart showing details of the main browser loop, including refreshing the applets.

[0017] Fig. 7 is a flow chart showing details of refreshing the applets in the main browser loop.

[0018] Fig. 8 is a flow chart showing a main JVM loop for regular non-reactive components.

[0019] Fig. 9 is a flow chart showing a main JVM loop for reactive components, such as a button press.

[0020] Fig. 10 shows further details of Figs. 8 and 9.

[0021] Fig. 11 is a diagram of the architecture of one embodiment of the present invention.

[0022] Figs. 12(a) and 12(b) are diagrams of, respectively, a pressed button and a released button.

[0023] Fig. 13 is a flow chart for orphan context.

[0024] Fig. 14 shows exemplary routines used to implement child and orphan contexts in an embodiment of the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

I. General Discussion

[0025] Many conventional World Wide Web browsers are capable of executing Java applets. A Java applet is a computer program written in the Java programming language that is designed to execute within a World Wide Web browser. Such applets are often started when a user clicking on a button or link in a displayed Web page. Once started, an applet executes inside the browser, performing whatever tasks it has been programmed to perform. "Java" is a trademark of Sun Microsystems, Inc. in the United States and other countries.

[0026] Fig. 1 (a) is a block diagram of a data processing system 100 in accordance with one embodiment of the present invention. Data processing system 100 can be, for example, a set top box 101 connected to a display device 130, such as, for example, a television set or some other appropriate display device. Data processing system 100 can also be, for example, any other appropriate data processing system.

[0027] System 100 includes a processor 102 and a data storage area (e.g., a memory) 104. Data storage area 104 includes certain well-known types of information, such as operating systems 112 and data and data structures 114. In the embodiment shown, data and data structures 114 include, for example, web pages capable of being displayed by a browser 106. Data and data structures 114 can also include, for example, applet software capable of being executed by a Java virtual machine (JVM) 108. Storage area 104 preferably also includes software (not shown) for communicating with a network connection 103, which can be a LAN, WAN, intranet, or the internet.

[0028] A person of ordinary skill in the art will understand that system 100 may also contain additional elements, such as input/output lines; additional or second input devices, such as a keyboard, a mouse, and a voice input device; and display devices, such as a display terminal. System 100 may also include a computer readable input device (not shown), such as a floppy disk drive, CD ROM reader, a chip connector, a chip connector, or a DVD reader, that reads computer instructions stored on a computer readable medium, such as a floppy disk, a CD ROM, a memory chip, or a DVD disk. System 100 also may include application programs, operating systems, data, etc., which are not shown in the figure for the sake of clarity.

[0029] In the following discussion, it will be understood that the steps of methods and flow charts herein discussed herein can be performed by processor 102 (or another appropriate processor) executing instructions stored in storage area 104 (or other appropriate memories or storage areas). It will also be understood that the invention is not limited to any particular implementation or programming technique and that the invention may be implemented using any appropriate techniques for implementing the functionality described herein. The invention is not limited to any particular programming language or operating system.

[0030] The instructions in storage area 104 may be read into storage area 104 from a computer-readable medium (not shown). Execution of sequences of instructions contained in main memory causes processor 102 (or a similar processor) to perform the processes described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

[0031] Fig. 1 (a) shows a virtual machine (such as a Java virtual machine) 108 executing on system 100. Fig. 1(b) is a block diagram of a virtual machine that is supported by system 100 of Fig. 1(a), and is suitable for implementing the present invention. When a computer program, e.g., a computer program written in the Java programming language, is executed, source code 160 is provided to a compiler 170 within compiletime environment 155. Compiler 170 translates source code 160 into bytecodes 180. In general, source code 160 is

translated into bytecodes 180 at the time source code 160 is created by a software developer.

[0032] Bytecodes 180 may generally be reproduced, downloaded, or otherwise distributed through a network, e.g., network 103 of Fig. 1(a), or stored on a storage device such as primary storage 104 of Fig. 1(a). In the described embodiment, bytecodes 180 are platform independent. That is, bytecodes 180 may be executed on substantially any computer system that is running on a suitable virtual machine.

[0033] Bytecodes 180 are provided to a runtime environment 108, which includes virtual machine 190. Runtime environment 108 may generally be executed using a processor or processors such as processor 102 of Fig. 1(a). Virtual machine 109 includes a compiler 192, an interpreter 194, and a runtime system 196. Bytecodes 180 may be provided either to compiler 192 or to interpreter 194.

[0034] When bytecodes 180 are provided to compiler 192, methods contained in bytecodes 180 are compiled into machine instructions. In one embodiment, compiler 192 is a just-in-time compiler, which delays the compilation of methods contained in bytecodes 180 until the methods are about to be executed. When bytecodes 180 are provided to interpreter 194, bytecodes 180 are read into interpreter 194 one bytecode at a time. Interpreter 194 then performs the operation defined by each bytecode as each bytecode is read into interpreter 194. That is, interpreter 194 "interprets" bytecodes 180, as will be appreciated by those skilled in the art.

[0035] When a method is invoked by another method, or is invoked from runtime environment 108, if the method is interpreted, runtime system 196 may obtain the method from runtime environment 108 in the form of sequences of bytecodes 180, which may be directly executed by interpreter 194. If, on the other hand, the method that is invoked is a compiled method that has not been compiled, runtime system 108 also obtains the method from runtime environment 108 in the form of a sequence of bytecodes 108, which then may go to activate compiler 192. Compiler 194 then generates machine instructions from bytecodes 180, and the resulting machine-language instructions may be executed directly by processor 102 (or other appropriate processors). In general, the machine-language instructions are discarded when virtual machine 109 terminates. The operation of virtual machines or, more particularly, Java virtual machines, is described in more detail in The Java Virtual Machine Specification by Time Lindholm and Frank Yellin (ISBN 0-201-63452-X), which is incorporated herein by reference.

[0036] Fig. 2 shows an example of a set top box system, including a set top box 101 connected to a television 130 and a remote unit 120. Remote unit 120, which is used as an input device, is not shown to scale. An actual remote unit is generally of a size appropriate to fit in a user's hand. Similarly, set top box 101 and television 120 are not necessarily shown to scale. Certain embodiments of the present invention can also be used with a keyboard or with an infrared or other wireless keyboard (not shown).

[0037] It will be understood that the button arrangement shown on remote unit 120 are for purposes of example only and are not to be interpreted in a limiting sense. Remote unit 120 includes a "web" button, a "mail button, a "home" button, a "goto" button, a "fave" button, a "back" button, a "ctl" button, arrow buttons, and a "select" button. As will be appreciated, a conventional use of back button is to display a previously displayed web page. Similarly, a conventional use of "goto" button is to allow the user to specify and view a new web page. The arrow buttons allow the user to move a cursor between various elements on web page 304, as is known to persons skilled in the art.

[0038] In Fig. 2, a button 204, a list and a panel are displayed on a web page displayed on display device 130. It will be understood that these elements 304 are generated on a web page under control of an applet (not shown), such as applet 110 of Fig. 1 (a).

[0039] Network connection 103 allows the set top box to communicate with a network, such as the internet, an intranet, a LAN or a WAN. Network connection can also be a wireless link.

[0040] Video line 207 provides a source of video input to set top box 101. Line 206 couples the set top box 101 to the display device 130 to convey video and audio output signals to the display device 130. The signals on line 206 could be, for example, one or more of, S-video signals, composite signals, component signals, or audio signals, HDTV signals.

II. Detailed Discussion

[0041] Fig. 3(a) is a block diagram of parent and child context data structures in accordance with an embodiment of the present invention. In Fig. 3(a), a browser page memory 312 includes memory for a first applet area (which contains a list, a button, and a panel) and a second applet area (contents not shown). The browser memory includes a pointer to a parent context data structure 314 for the browser, which includes information about the browser, such as font, foreground and background color information, and screen coordinates of the browser. The parent context 314 has an associated parent memory buffer 318, which includes visual representations of the screen memory for the browser.

[0042] Each element in the HTML for a page currently being displayed by the browser has an associated browser element 316. Thus, in the example, each of the first applet and the second applet have an associated browser element 316. Other types of elements that may have an associated browser element include GIFs, JPEGs, etc.

[0043] Each browser element points to a child context 302 in the JVM. Only one child context 302 is shown in the figure for the sake of clarity, but in the described em-

bodiment, each browser element (and therefore each applet) has an associated child context. The child context 302 includes information about its associated applet, such as font, foreground and background color information, and screen coordinates of information displayed by the applet. The child context of an applet also includes a damage flag 350, which indicates whether the contents of the child buffer have been changed (akin to a "dirty bit) and a Recursive Update flag 352, which indicates whether the applet contains a reactive element, such as a button, list, or choice. The child context 314 has an associated child memory buffer 304, which includes visual representations of the screen memory for the browser.

[0044] The described embodiment does not use a graphics system such as an X Server or Motif. Instead, it uses direct graphic routines implemented along with child and parent contexts. These direct graphics routines write directly into the graphics buffers of the parent and/or child contexts. A graphics Java object 310 points to a graphics peer object 308, which is an abstract class. A DWL object is a subclass of class 308. An example of the DWL graphics object could be a button widget. The DWL graphics object points to the child context 302 for the applet that contains the object.

[0045] It is important to note that each Java applet executes as one or more separate threads in the JVM. Thus, each Java applet can modify its child context 302 and its child memory independently from the other applets. As shown in Fig. 6, a main browser loop periodically polls the executing applets to determine whether any of their damage flags are set.

[0046] Fig. 3(b) is a diagram showing the relationships between a browser, exemplary browser elements, and exemplary components of a specific browser element. A browser displays HTML elements such as applets, GIFs, JPEGs and frames. Certain elements, such as applets can include graphics components such as buttons, lists, choices, or panels. As shown, certain elements such as buttons, lists, and choices are examples of reactive components that require an interactive output when they are pressed or selected by the user. In contrast, panels generally do not require any external indication that the user has used them.

[0047] Fig. 4 is a block diagram of an orphan data structure in accordance with an embodiment of the present invention. Orphan contexts are used in situations where some "behind the scenes" drawing needs tot be done. An orphan context is similar to a child context (and can be a special case of a child context) but is not associated with any particular applet or other element. Instead, as shown in Fig. 13, the content of an orphan context is transferred to a child buffer and the damage flag of the child buffer set so that the main browser loop will eventually pick up the graphics change. Orphan contexts cannot be drawn directly on the screen. They can only be copied to another context (such as a child context).

[0048] Fig. 5 is a block diagram showing a main browser loop and a JVM loop. As is shown in the Figure, these loops include a main browser loop (which is performed by browser 106 of Fig. 1). A main JVM loop 504 is performed by JVM 108. A key event queue 510 receives "key events," for example, key presses from remote unit 206 and queues them for input to main browser loop 502. A JVM queue 508 receives "JVM events," from the main browser loop and queues them for input to main JVM loop 504. In the described embodiment, each of loops 502and 506 are implemented as a thread executed by processor 102.

[0049] Fig. 6 is a flowchart showing details of the main browser loop, including periodic refreshing of executing applets. In step 602, the main loop receives a key press from a key event queue. A key press can be the result of the user pressing a key on the remote 120 or the result of clicking or selecting a button or similar component displayed on the screen by an applet. If in step 602, the current key event is an event for an applet, an event is placed on the JVM queue 508. Otherwise, an appropriate activity is performed for the key event. In either case, the main browser loop refreshes the graphics display of any executing applets, as shown in Fig. 7. (It will be appreciated that other activities occur in the main browser loop, but have been omitted from the description for the sake of clarity).

[0050] Fig. 7 is a flow chart showing details of refreshing the applets in the main browser loop. In an initial step 702, control waits until the parent context is unlocked. In some situations, the parent context will already be unlocked. Locking of the parent context occurs when some other thread needs to access the parent context or wants to keep the parent context from being accessed. In the described embodiment, this lock is implemented as a unix mutex lock. One example of how the parent might become locked is shown in step 1002 of Fig. 10.

[0051] The steps of Fig. 7 loop through the active applets to determine whether its damage flag is set. If, in step 704, the damage flag of a current applet is set, control passes to step 706. Otherwise, control passes to step 712. In step 706, the browser locks the parent and child contexts (so not other threads can change them during steps 708-710). In step 708, the browser "pulls" graphics data from the child context 302 and child memory 304 of the applet to the parent context 314 and parent memory 318 of the browser. In the described embodiment, this step is performed via a call to a routine called aglPutContext, which is described in connection with Fig. 14.

[0052] Fig. 8 is a flow chart showing a main JVM loop for regular non-reactive components, such as a panel, or other display element that does not require an interactive behavior. In the described embodiment, the steps of Fig. 8 are performed by the JVM 108. In step 802, the JVM receives an event from the JVM event queue. Events in this queue can be sent from the browser or from some other source. An example of such a non-re-

active event might be "draw a panel." Steps 804-812 locate a Java AWT widget that has focus (i.e., that has been chosen to receive the input from remote 120 or a similar keyboard). The event is delivered to a Java widget, which invokes the abstract event handler code and the actual component code for the peer. Step 812 locates the DWL component widget corresponding to the peer, and, in step 814, the event is delivered to the DWL (Developer's widget library) component (see Fig. 14). In step 816, the DWL component draws the graphic in the child buffer in accordance with the event. This step is described in more detail in Fig. 10, which describes a method used for both reactive and non-reactive components.

[0053] Fig. 9 is a flow chart showing a main JVM loop for reactive components, such as a button press. As discussed above, a reactive component cannot wait for the main browser loop. In the described embodiment, the main browser loop of Fig. 7 executes every 200 milliseconds. Similarly, the described reactive events, such as a button press are required to take place in approximately 50 milliseconds. The steps of Fig. 9 are similar to those of Fig. 8, except that, in step 918, the JVM twice writes directly into the child and the parent contexts. Thus, the JVM directly writes the "pressed" button graphic and the "released" button graphic without waiting for the main browser loop. Step 918 sets the Recursive Update flag, draws two graphics images into the child and parent buffers, and clears the Recursive Update flag. This is described in more detail in connection with Fig. 10.

[0054] Fig. 10 shows further details of Figs. 8 and 9. If the method of Fig. 10 is called from Fig. 8 (a non-reactive component), the Recursive Update flag will not be set. If, on the other hand, the method of Fig. 10 is called from Fig. 9 (a reactive component), the Recursive Update flag will have been set in step 918.

[0055] Thus, for a non-reactive component, in step 1002 the widget issues an aglBeginPaint call (see Fig. 14), which sets the damage flag in the child context. Because the recursive update flag is not set, control passes to step 1004. In step 1004, the DWL component object draws a graphics for the received JVM event in the child buffer. Because the recursive update flag is not set, control passes to step 1006 and then to step 1008. Thus, for a non-reactive component, the component object draws the correct graphic in the child buffer and sets the damage flag in the child context. The change in graphics will be picked up in the main browser loop as shown in Fig. 7.

[0056] For a reactive component, in step 1002 the widget issues an aglBeginPaint call (see Fig. 14), which sets the damage flag in the child context (even though this is not required, it is harmless). Because the recursive update flag is set, a lock is also paced on the parent context and buffer and on the child context and buffer. Such a lock means that only the locking thread can change the contexts until they are unlocked. Control passes to step 1004. In step 1004, the DWL component object draws a first graphic for the received JVM event in the child buffer (such as a "pressed" button shown in Fig. 12(a)). Because the recursive update flag is set, the DWL component object also draws the graphics for the received JVM event in the parent buffer. Thus, the change to the parent buffer will automatically be reflected on the display screen without having to wait for the main browser loop. Control passes to step 1006 and then to step 1009, which clears the locks. Step 1010 waits some approximate time, such as 50 milliseconds, which is long enough for a human user to notice that the button appearance has changed.

[0057] Steps 1012-1016 perform similar steps to draw a second graphic (such as a "released" button shown in Fig. 12(b)) directly into the parent and child buffers and contexts. Thus, for a reactive component, the component object draws a first graphic in the child and parent buffers, waits some predetermined period of time, and draws a second graphics in the parent and child buffers. Reactive graphics do not wait for the change in graphics will be picked up in the main browser loop as shown in Fig. 7. If the main browser loop occasionally happens to redraw the second graphic, no harm is done.

[0058] Fig. 11 is a diagram of the architecture of one embodiment of the present invention. It includes the extended agl and DWL routines mentioned above.

[0059] Fig. 13 is a flow chart for orphan context. The steps are similar to those of Fig. 8, but in step 1316, the orphan context must be copied to a child context (or a parent context, not shown) where it will eventually be displayed. Orphan context are used, for example, when there is a need to draw a graphic "offscreen," such as sometimes occurs for JPEGs and GIFs. Similarly, if a graphic will be duplicated once it is drawn, it is often advantageous to use an orphan context.

[0060] Fig. 14 shows exemplary agl (applications graphics library) routines used to implement child and orphan contexts in an embodiment of the present invention. These routines include but are not limited to: aglCreateChildContext, aglDeleteChildContext, aglBeginPaint, aglEndPaint, aglCreateOrphanContext, aglPutContext, and aglContextRedrawn.

[0061] In summary, the described embodiment of the present invention provides one or more child contexts that correspond to elements in the HTML for a web page displayed by a browser. For example, each applet executed by the browser has a corresponding and separate child context. Each child context has an associated memory buffer. The browser also has a parent context, which each child context points to. When a graphic is displayed via a widget, the widget draws the graphic (such as a panel or a non-pressed button) in the child context of the applet and sets a "damage" flag in the child context. When the browser performs its main browser loop, it checks the status of the damaged flag for each element (including each applet). If the browser finds a damaged flag that is set, it "pulls" the information

from the child buffer into the parent buffer, which is used to update the display screen. Thus, several separate threads in the JVM can be executing concurrently and updating graphics elements without interfering with each other.

[0062] The invention extends the AGL (Applications graphics library) routines to include (at least) routines to set up, delete, read, and modify child contexts and to copy information from child contexts to parent contexts. The DWL (developer's widget library) routines are extended to include a parameter identifying the context currently being worked on. The browser has been extended to implement the functionality of checking the damage flag of the context for each applet and pulling information from a buffer of a child context into the buffer of the parent context.

[0063] Reactive components, such as buttons, lists, choice bars, etc. present special problems. When a widget, for example, a button, needs to show interactive behavior when pressed (like going up and down), this behavior must occur very rapidly - too rapidly to necessarily be picked up by a main browser loop, which executes relatively infrequently. Thus, a child context cannot update the display properly for a reactive component if it merely marks the child context as damaged and waits for the browser loop.

[0064] While the invention has been described in conjunction with a specific embodiment, it is evident that many alternatives, modifications and variations will be apparent to those skilled in the art in light of the foregoing description. Accordingly, it is intended to embrace all such alternatives, modifications and variations as fall within the spirit and scope of the appended claims and equivalents.

Claims

1. A method of displaying a reactive graphics element, comprising:

   sending a reactive graphics event, by a browser to a virtual execution environment;
   providing, in the virtual execution environment, a child context corresponding to a parent context in the browserand an applet that is executed by the browser;
   writing, by the virtual execution environment, a first graphic into a buffer of the child context and into a buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen;
   waiting a predetermined amount of time;
   writing, by the virtual execution environment, a second graphic into the buffer of the child context and the buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen; and
   returning execution control to the browser from the virtual execution environment.

2. The method of claim 1, further comprising providing a second child context in the virtual execution environment, corresponding to the parent context in the browser, where the child context corresponds to a second applet that is executed by the browser.

3. An apparatus that displays a reactive graphics element, comprising:

   a browser having a parent context used to update a display, the browser being configured to send a reactive graphics event to a virtual execution environment;
   software circuitry configured to provide, in the virtual execution environment, a child context corresponding to the parent context and an applet that is executed by the browser;
   software circuitry configured to write, by the virtual execution environment, a first graphic into a buffer of the child context and a buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on the display ;
   software circuitry configured to wait a predetermined amount of time;
   software circuitry configured to write, by the virtual execution environment, a second graphic into the buffer of the child context and the buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on the display; and
   software circuitry configured to return execution control to the browser from the virtual execution environment.

4. A computer program product comprising:

   a computer usable medium having computer readable code embodied therein for causing a data processing system to display a reactive graphics element, including:
   computer readable program code configured to cause the data processing system to send a reactive graphics event, by a browser to a virtual execution environment;
   computer readable program code configured to cause the data processing system to provide a child context in the virtual execution environment, the child context being associated with a parent context in the browser, where the child context corresponds to an applet executed by the browser;

      computer readable program code configured to cause the data processing system

to write, by the virtual execution environment, a first graphic into a buffer of the child context and a buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen;
computer readable program code configured to cause the data processing system to wait a predetermined amount of time;
computer readable program code configured to cause the data processing system to write, by the virtual execution environment, a second graphic into the buffer of the child context and the buffer of the parent context, in accordance with the reactive graphics event, which will be displayed on a display screen; and
computer readable program code configured to cause the data processing system to return execution control to the browser from the virtual execution environment.

5. The method of claim 1, wherein the virtual execution environment provides applications graphics library (AGL) routines, including a child context set-up routine, a child context delete routine, a child context read routine, a child context modify routine, and a routine for copying information from the child context to the parent context.

6. The method of claim 5, wherein the method further includes invoking a routine, the routine being one of the AGL routines.

7. The method of claim 1, wherein the applet is one of multiple applets executed as different threads by the browser, each of the applets being associated with a child context, each child context corresponding to the parent context.

8. The method of claim 7, wherein each child context corresponds to a browser element displayed on the display.

9. The method of claim 1, wherein the child context further includes a damage flag indicating whether the contents of a corresponding child buffer has changed.

10. The method of claim 1, wherein the child context further includes a recursive update flag indicating whether the associated applet contains a reactive element.

11. A method of graphic widget interaction, comprising:

receiving a key event indicating one of a press, release and selection, the key event being either reactive or non-reactive, the selection indicating selection of an item displayed by an applet that is executed by a browser, the browser having a parent context that is used for updating a display;
providing, in a virtual machine, a child context associated with the parent context; and
performing, by the virtual machine, a virtual machine loop, including:

locating a graphic widget corresponding to the key event;
setting a recursive update flag in the child context if the key event is reactive;
resetting the recursive update flag in the child context if the key event is non-reactive;
setting a damage flag in the child context;
using the corresponding graphic widget to update the child context alone if the recursive update flag is reset; and
updating both the child context and the parent context, via the corresponding graphic widget, if the recursive updating flag is set, wherein the updating of the parent context is reflected automatically on the display without having to wait for a browser loop.

12. The method of claim 11, further including:

placing the key events in a queue for input to a browser loop; and
performing, by the browser, the browser loop, including;

performing an activity if the key event is press or release;
posting a virtual machine event in a queue for input to the virtual machine loop, if the key event is selection; and
refreshing each applet that is executed by the browser, the refreshing for each applet including:

using the damage flag for determining the need for refreshing the applet;
locking an associated child context and the parent context; and
pulling the graphics data from the associated child context to the parent context for updating the display.

13. The method of claim 12, wherein the key events of press and release indicate pressing and releasing, respectively, of one of a button and key.

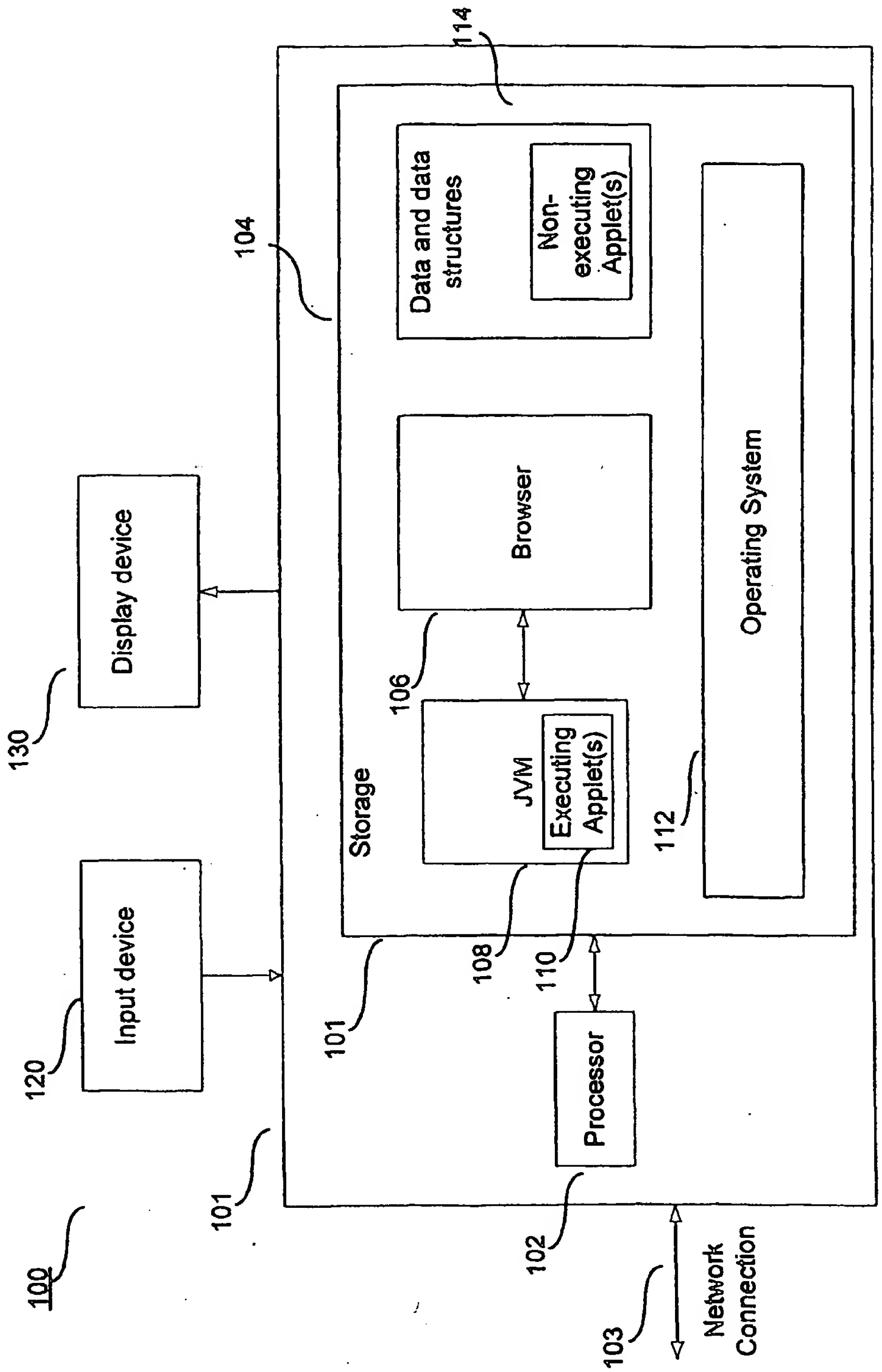

100
101
101
120
Input device
130
Display device
104
114
Storage
Data and data structures
Non-executing Applet(s)
Browser
106
JVM
Executing Applet(s)
108
110
112
Operating System
Processor
102
103
Network Connection

Fig. 1(a)

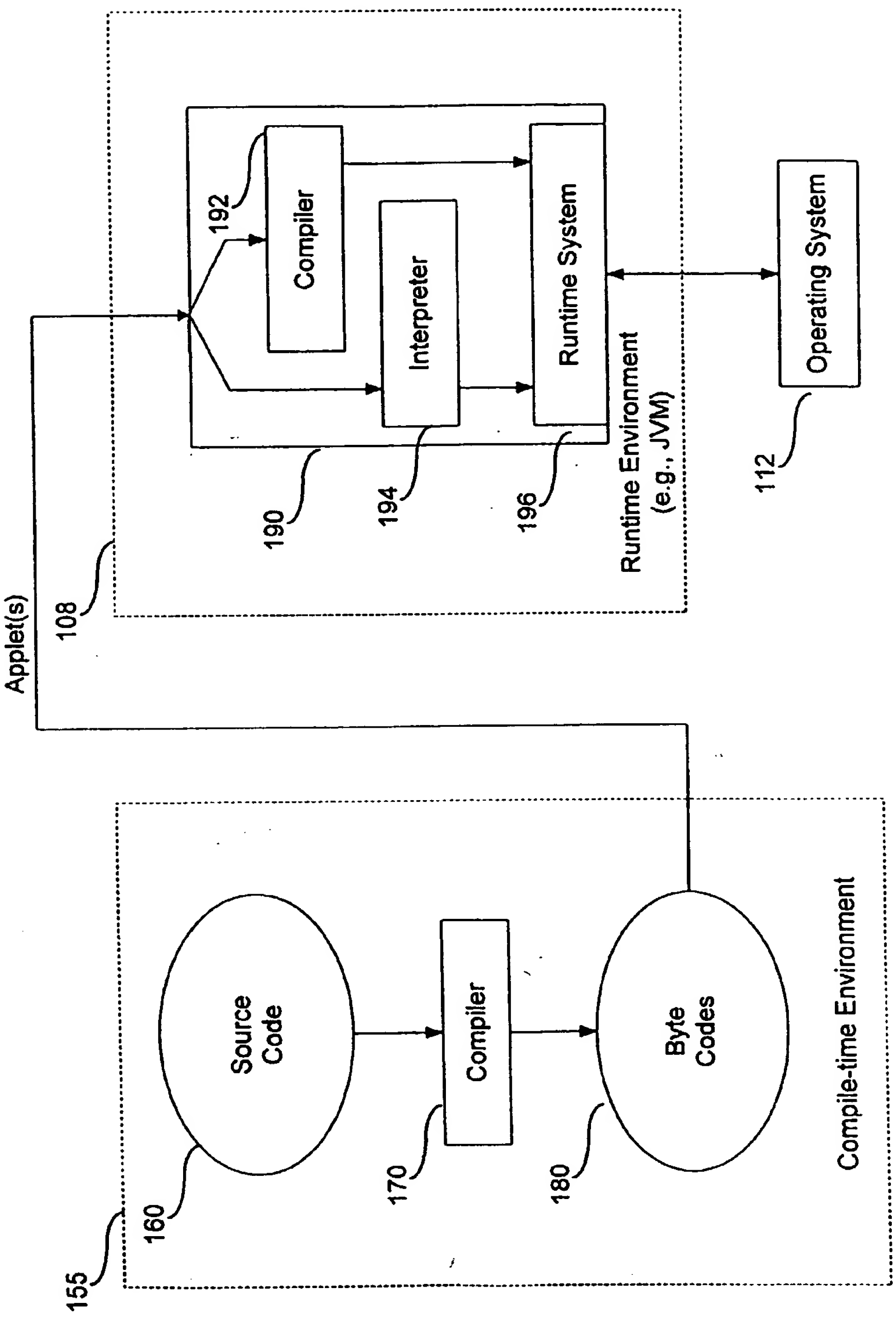

Compiler
Interpreter
Runtime System
Operating System
192
190
194
196
112
Runtime Environment
(e.g., JVM)
Applet(s)
108
Source
Code
Compiler
Byte
Codes
Compile-time Environment
155
160
170
180

Fig. 1(b)

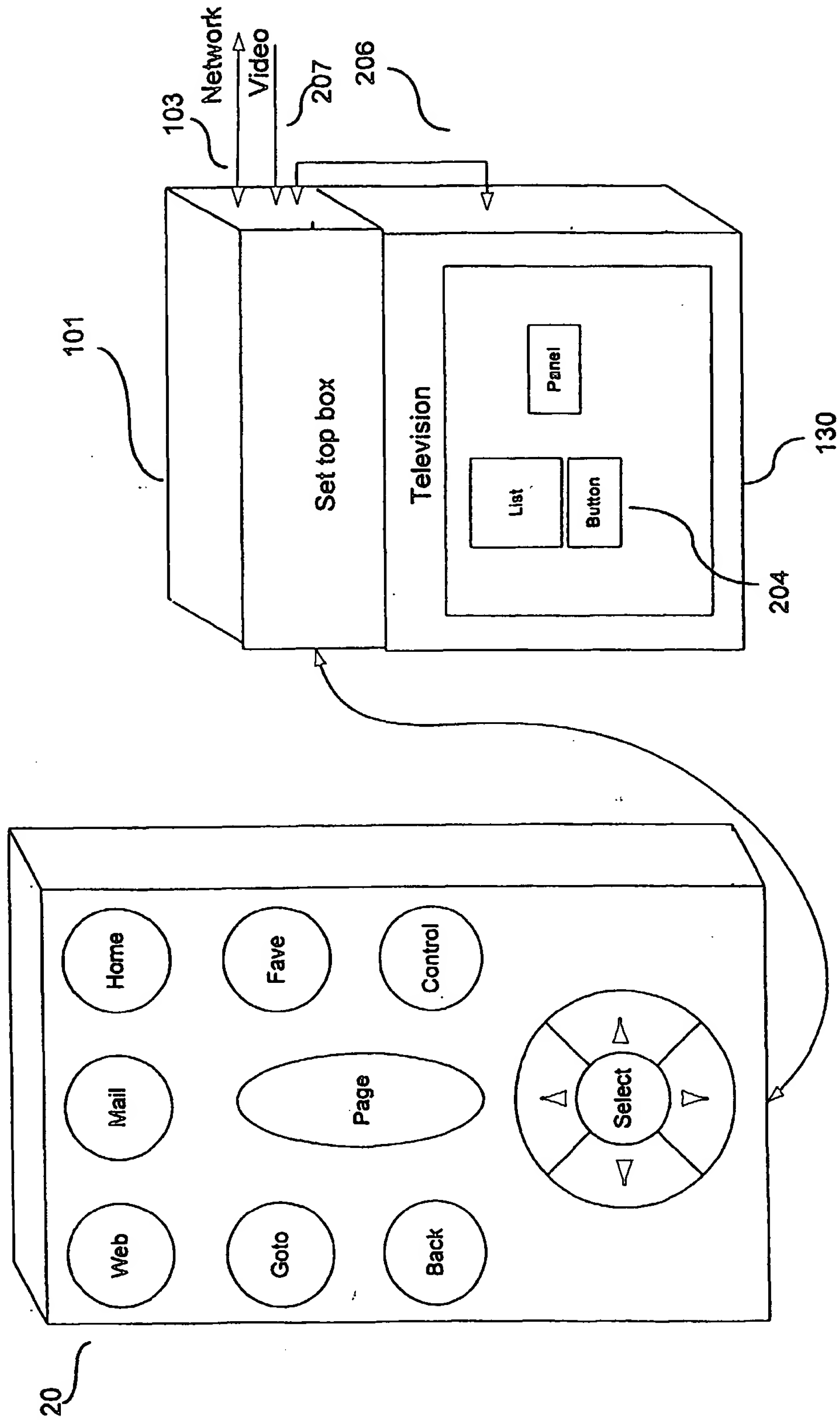

103
Network
Video
207
206
101
Set top box
Television
Panel
List
Button
130
204
120
Home
Fave
Control
Mail
Page
Select
Web
Goto
Back

Fig. 2

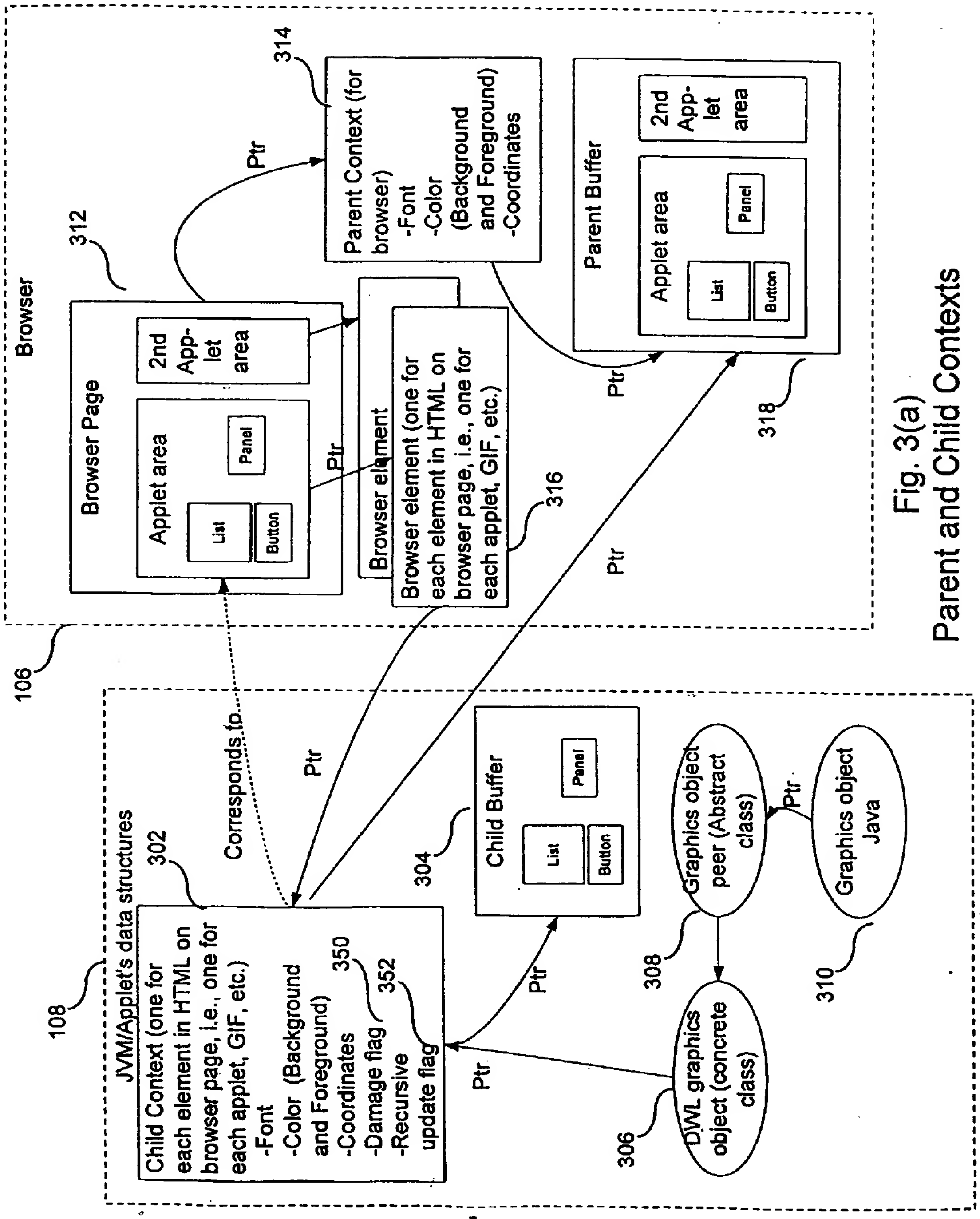


Fig. 3(a)
Parent and Child Contexts

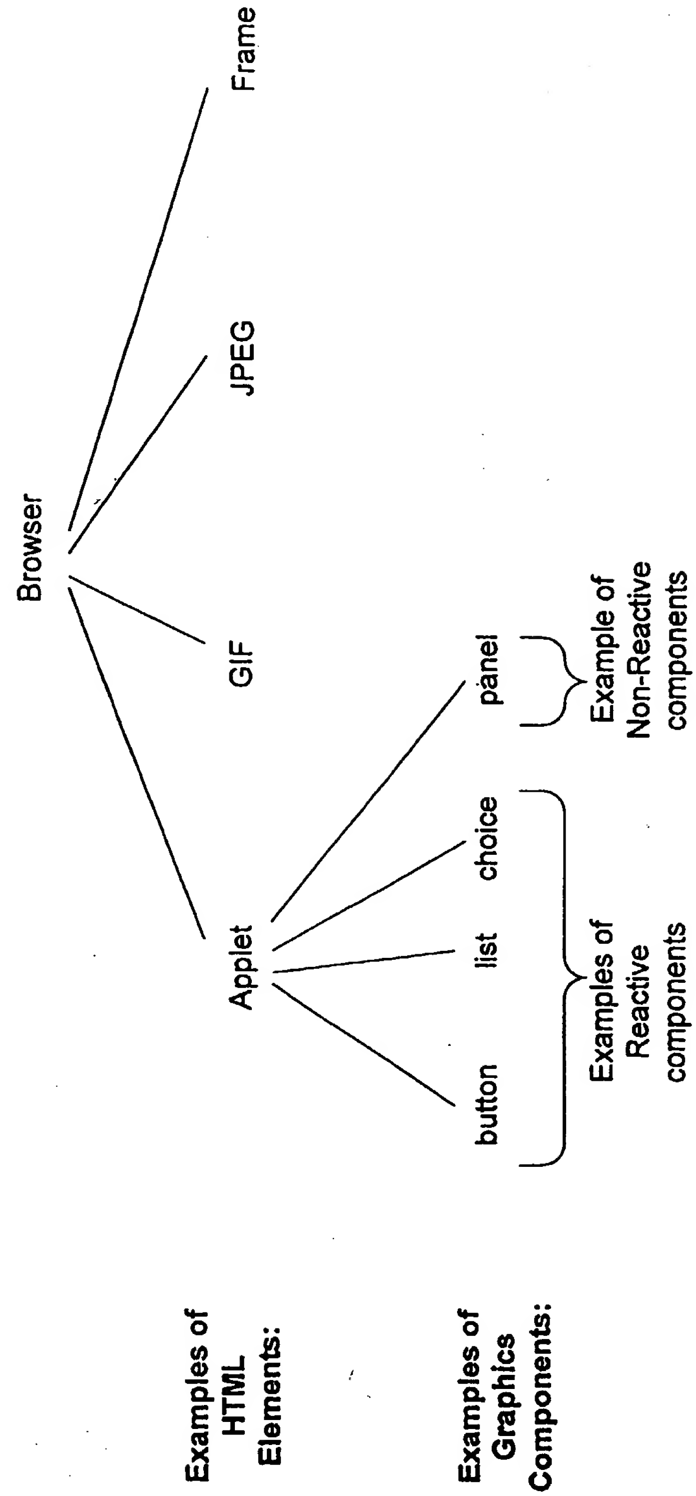

Browser
Frame
JPEG
GIF
Applet
button
list
choice
panel
Examples of Reactive components
Example of Non-Reactive components
Examples of HTML Elements:
Examples of Graphics Components:

Fig. 3(b)

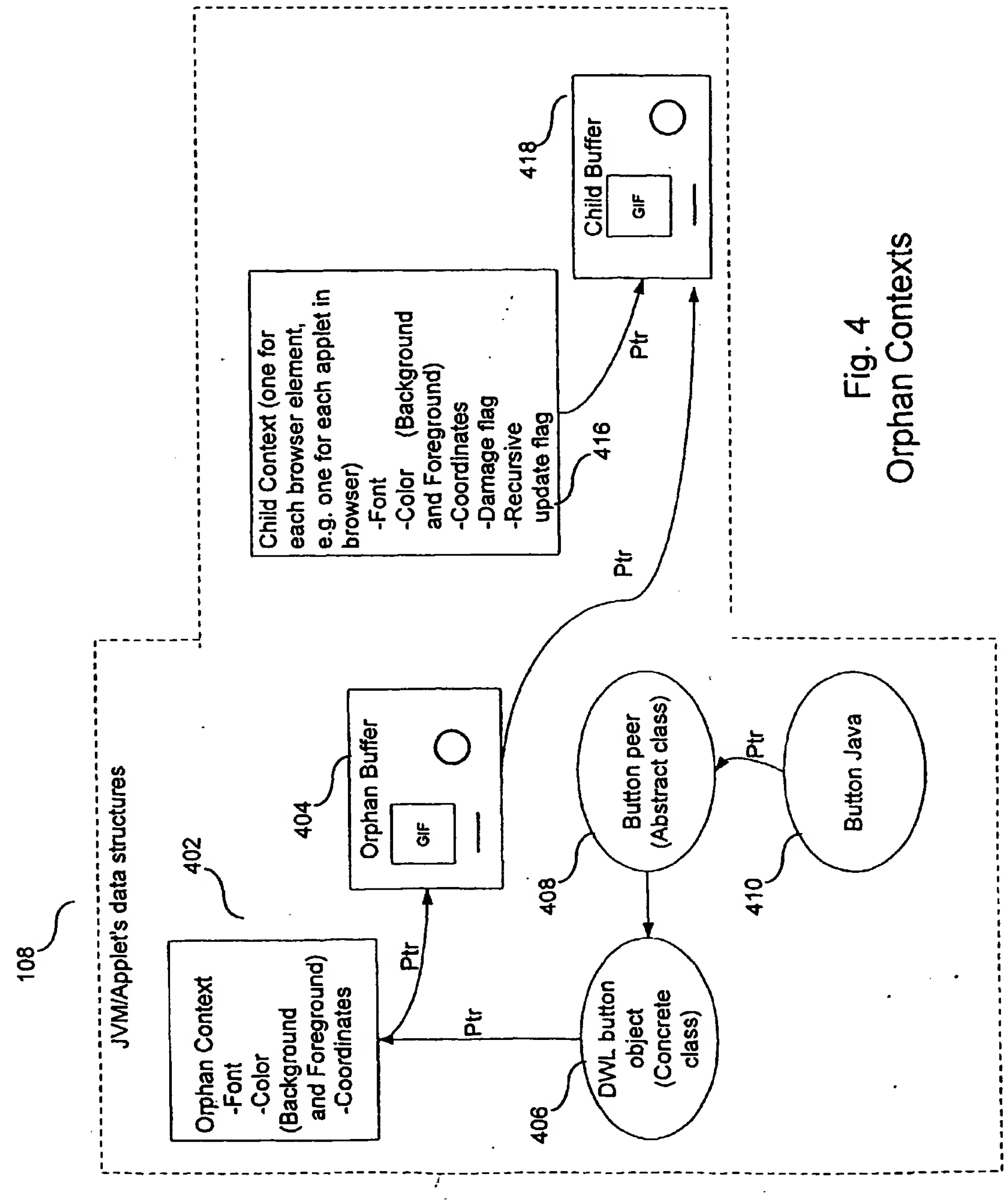
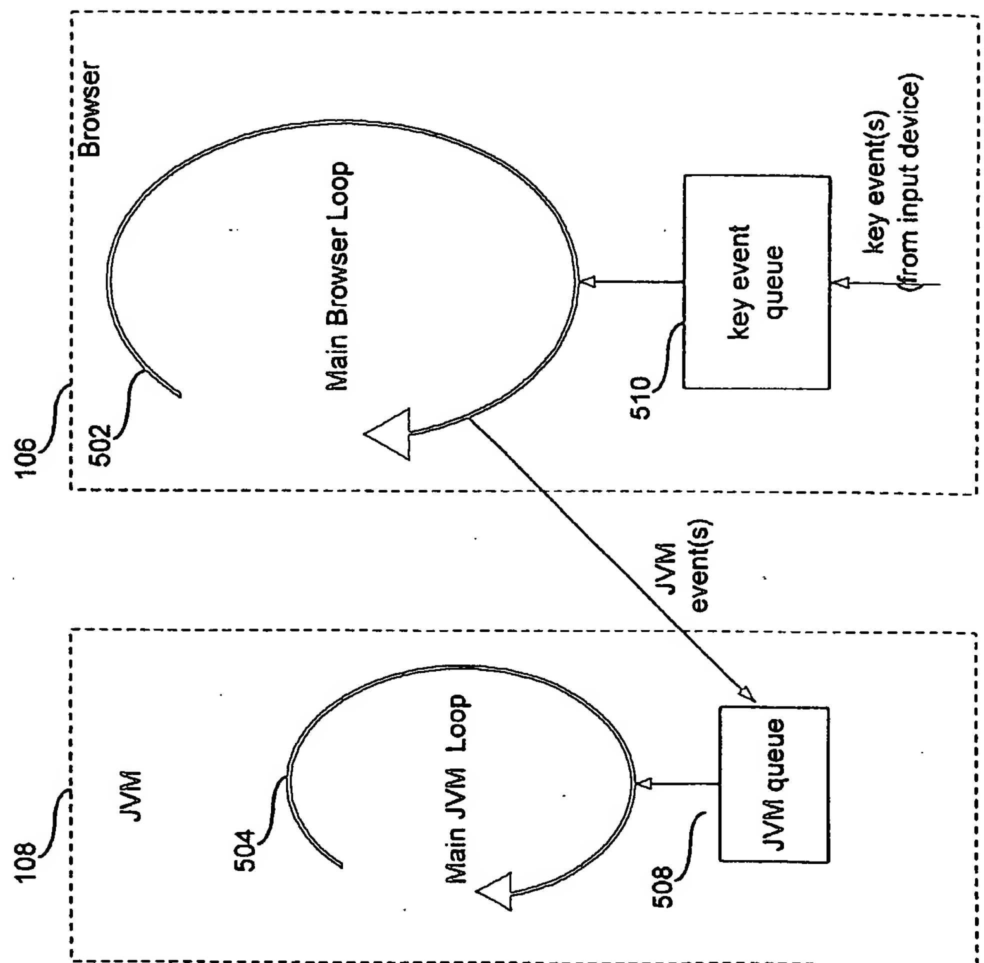


Fig. 4
Orphan Contexts



106
Browser
502
Main Browser Loop
510
key event queue
key event(s) (from input device)
JVM event(s)
108
JVM
504
Main JVM Loop
508
JVM queue

Fig. 5

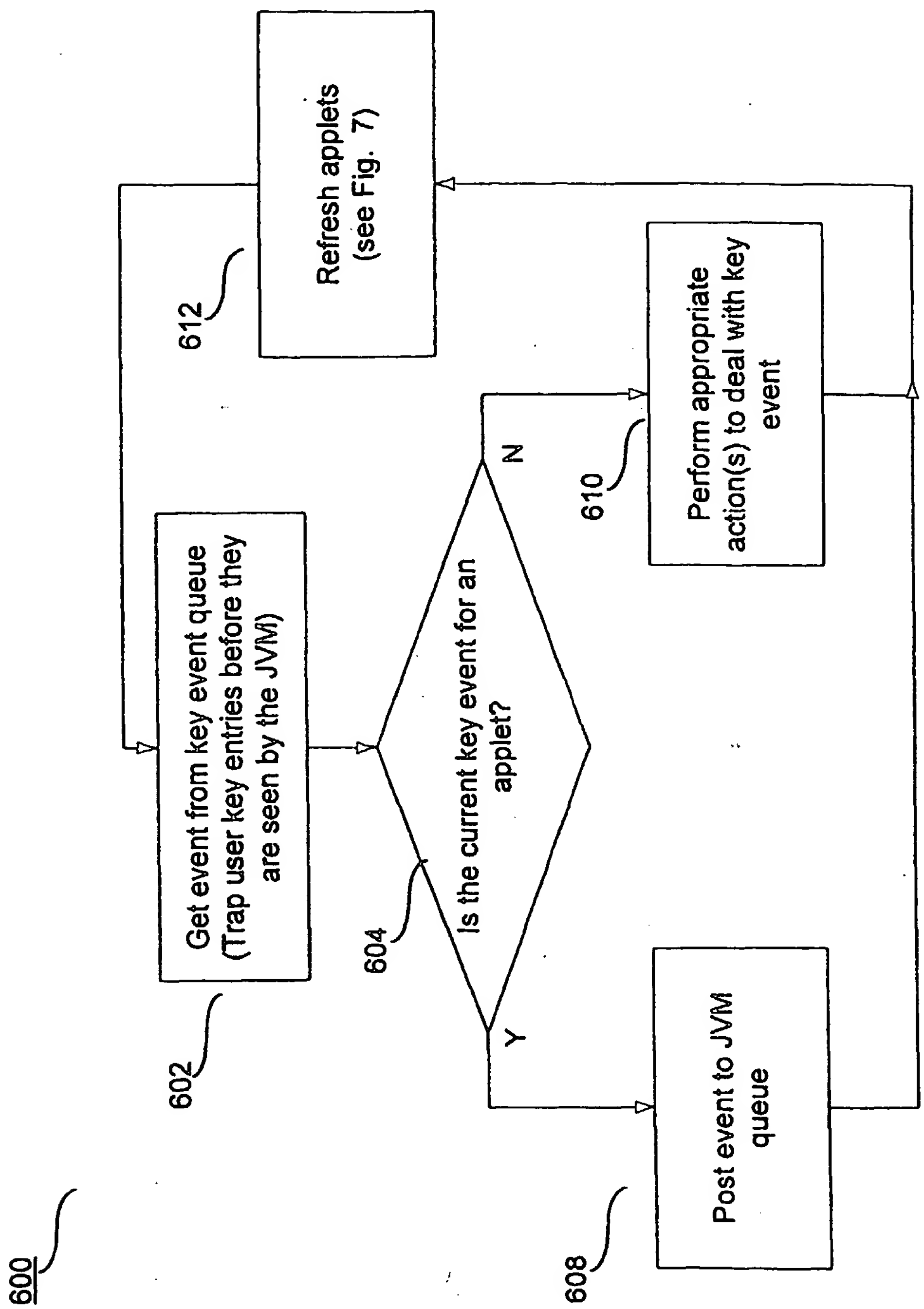
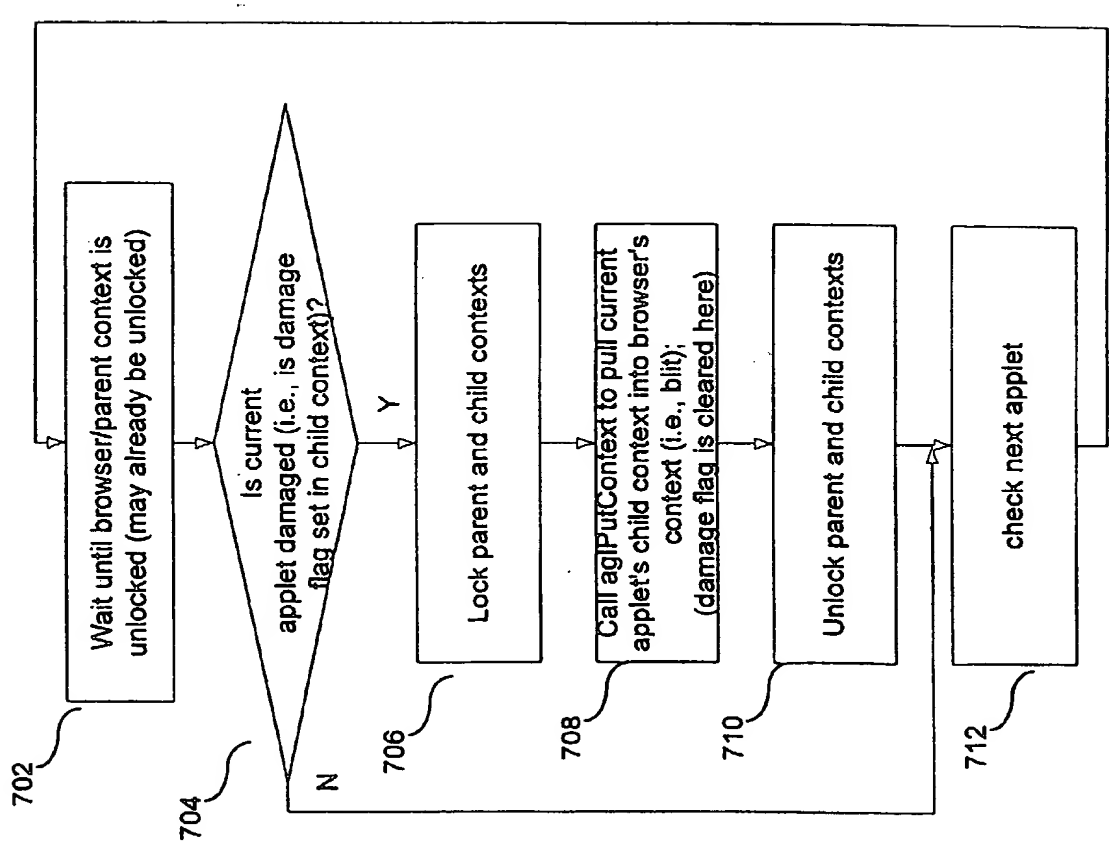


Fig. 6
Main Browser Loop

Fig. 7
Refresh Applets (in main Browser Loop)



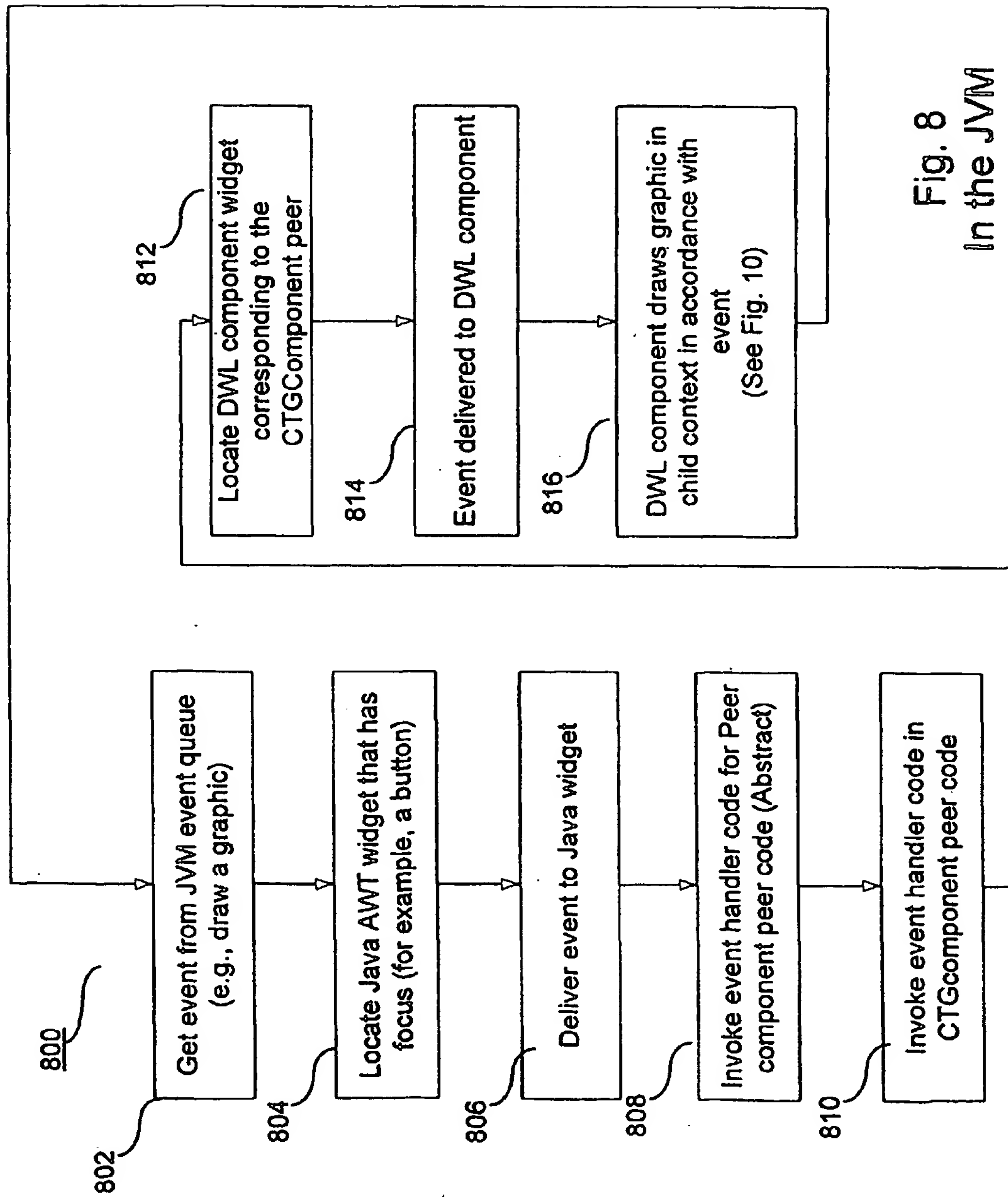


Fig. 8
In the JVM
(General case)

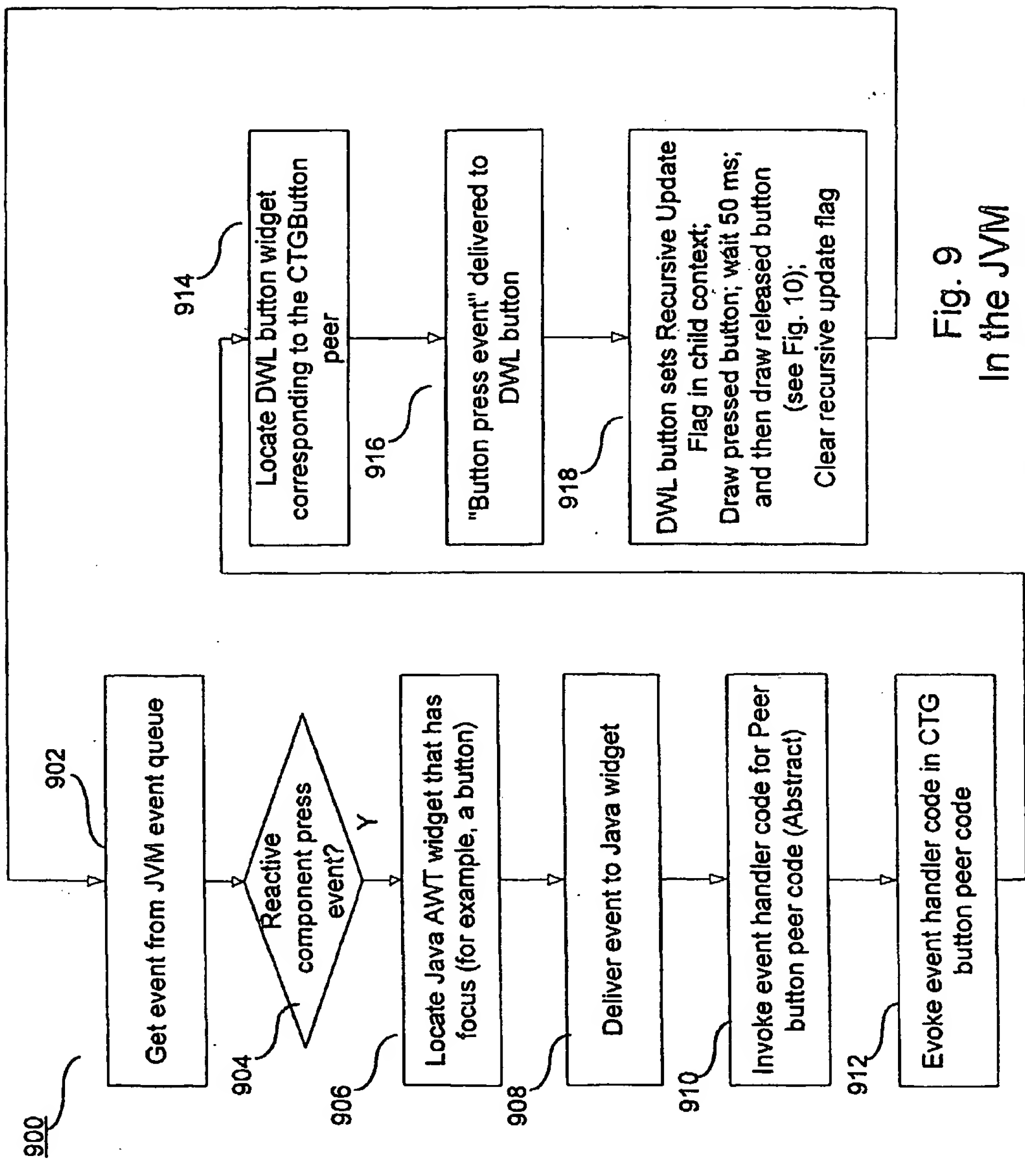


Fig. 9
In the JVM
(Button press/example of reactive component press)

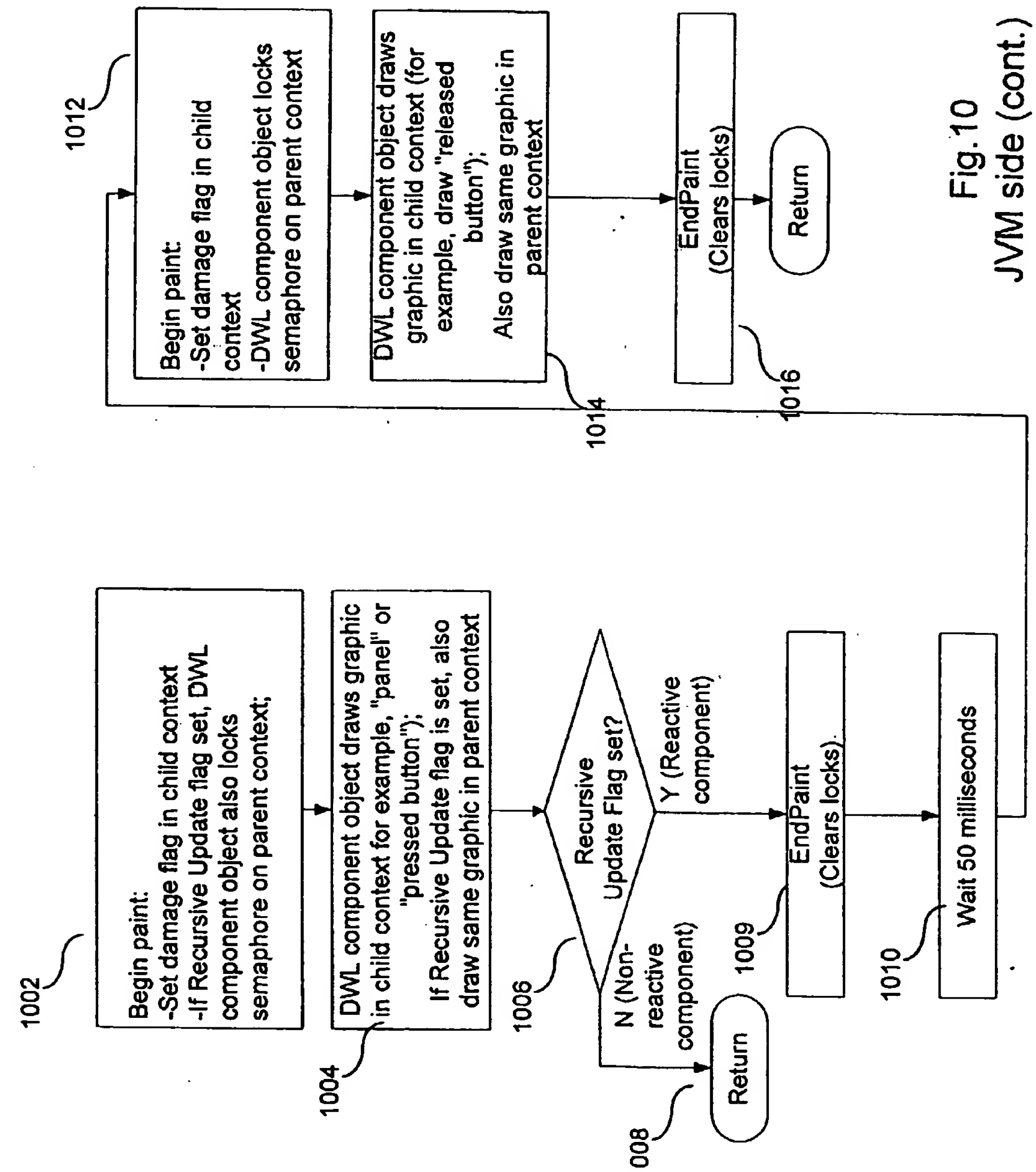
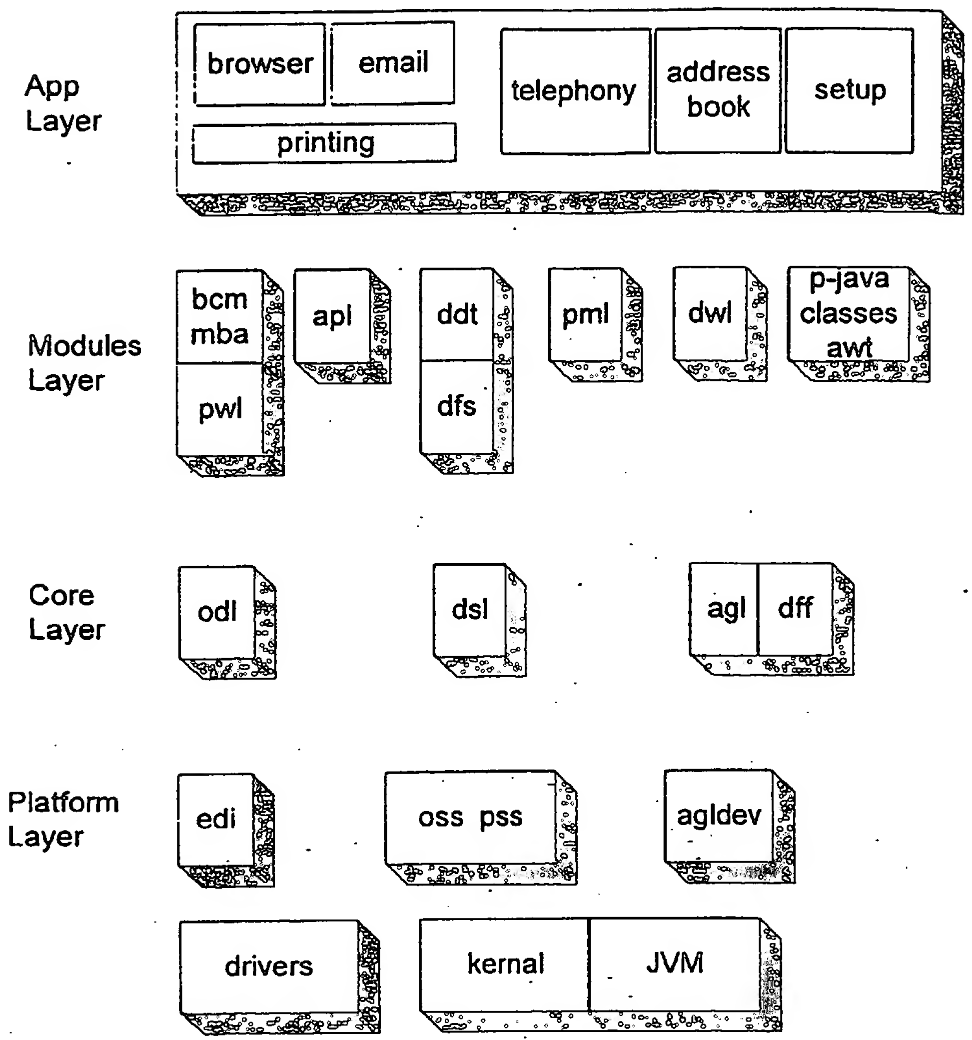


Fig.10
JVM side (cont.)

CED Consumer Device Architecture



BCM: Browser Control Module
MBA: Merrimack Browser A
DWL: Developer's Widget Library
ODL: Opaque Device Library
DSL: Developer's System Library
AGL: Applications Graphics Lab
DFL: Developer Font Library
ODI: Opaque Device Implementation
OSS: O S Specific
PSS: Project Specific Services

APL: Appliance Printer Library
PML: Programmers Mail Library
DFS: Developer File S
PWL: Programmer's WEB Library

Fig. 11

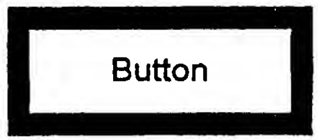


Fig. 12(a)
Graphic for
pressed button
(displayed for 50
milliseconds)

Fig. 12(b)
Graphic for
released button

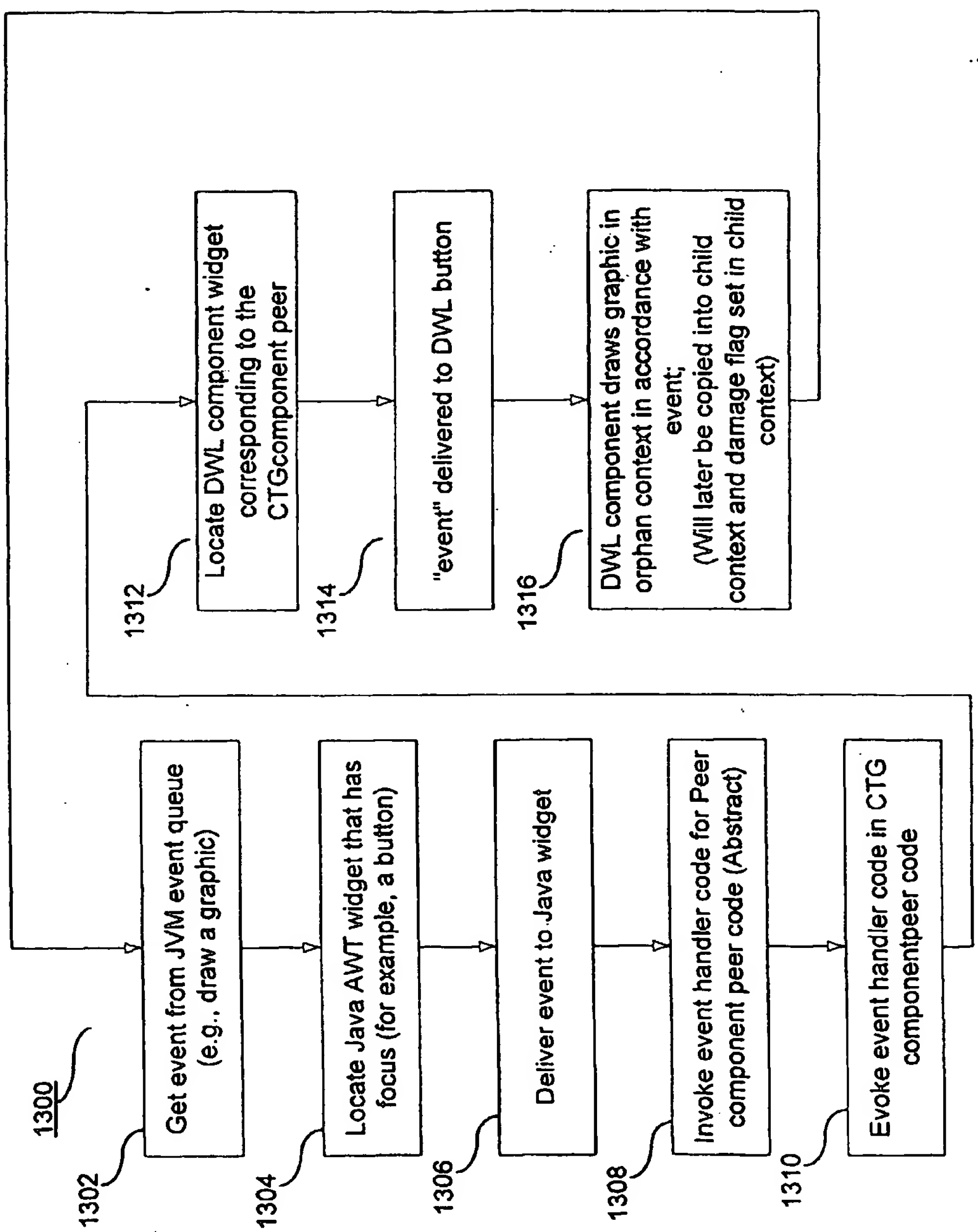


Fig. 13
In the JVM (Orphan contexts)

<u>aglCreateChildContext</u>
General description: Creates a child data structure
Arguments: parent context - parent context
rect - position and size of the child context relative to the parent
Returns: valid pointer to created context

<u>aglDeleteChildContext</u>
General description: Deletes a child data structure
Arguments: context - the context to be deleted
Returns: 0 - operation was immediately effective
1 - operation scheduled after threads are done with context
-1 - failure

<u>aglBeginPaint</u>
General description: Tell AGL that it is ready to begin drawing in the context
Arguments: context - the context to be drawn in
Returns: 0 - OK
-1 - failure

<u>aglEndPaint</u>
General description: Tell AGL that it is ready to be drawn by the screen
Arguments: context - the child context to be drawn to the screen
Returns: 0 - OK
-1 - failure

<u>aglCreateOrphanContext</u>
General description: Creates an orphan child context, which cannot be drawn directly to the screen. It can only be copied into another child context using algPutContext
Arguments: width
height
Returns: valid pointer to created context

<u>aglPutContext</u>
General description: Copies a child context to another at the position given by rect
Arguments: Target context
Source context
Rect - position on the target where the image is copied from

<u>aglContext Redrawn</u>
General description: Checks to see if a browser component (such as an applet) needs to be redrawn (i.e., whether damage flag is set)
Arguments: context - the context to check
Returns: TRUE - Applet need to be redrawn
FALSE - Applet does not need to be redrawn

Fig. 14
Example AGL calls